Kamila Szewczyk

An introduction to functional and array programming

2021 - 2023

Abstract

This book, which means to serve as an introduction to functional and array programming utilises KamilaLisp, a Lisp-inspired dynamically typed programming language that borrows many ideas from Haskell, Standard ML, APL, Scheme and others. It is a functional programming language with a strong emphasis on array programming, and it is designed to be used in a wide range of applications. Its design stems from previous iterations, which featured more orthodox Lisp syntax, lazy evaluation and symbolic computation with an emphasis on mathematical programming.

Foreword

KamilaLisp, the language described in the book, has its origins in its previous iterations - v0.1 and MalbolgeLISP. The first iteration of MalbolgeLISP (v1.0) was released in August 2020 and had very few of the features that distinguish KamilaLisp today. The second iteration of MalbolgeLISP (v1.1) was released in July 2021 and was the subject of much attention due to its unusual choice of implementation language. The final MalbolgeLISP (v1.2) was released in September 2021, sharing many ideas and features with KamilaLisp. Unfortunately, the effort required to implement it continued to grow. The codebase was becoming increasingly disorganised and difficult to maintain. In addition, the implementation of new complicated features was increasingly difficult due to the choice of implementation language. The first version of KamilaLisp, a MalbolgeLISP-inspired Lisp dialect, appeared in December 2021, featuring lazy evaluation, sophisticated numerical and symbolic operations, and many original design choices borrowed from MalbolgeLISP. Progress on the language has stalled due to significant performance problems with the Java Virtual Machine, the overhead of lazy evaluation, and some questionable decisions made at the core of the interpreter. After realising this rather early, an effort to create a C++ rewrite of the original code base began towards the end of 2021, but was quickly abandoned due to problems with garbage collection, efficient memory management, and C++ lacking certain qualities that would otherwise make it a good language in which to implement an interpreter for KamilaLisp. In August 2022 I started working on KamilaLisp v0.2, a Java rewrite of the original codebase that does not have some of the problematic features of the previous versions. This book, in the hope of being useful to the determined reader who wants to learn about KamilaLisp, describes my experiences in implementing a complex language runtime using Java, and also describes the language itself. It is beneficial if the reader has some knowledge of the APL programming language, however it is not necessary to understand the book to its fullest.

Contents

Glossary

In this book, the following assumptions are made:

- *Folding* a list `⍵` with a function `⍺⍺` is equivalent to putting `⍺⍺` between every element of `⍵`, also given the identity element of `⍺⍺`. One could assume binding to the left (`(⍵[1] ⍺⍺ ⍵[2]) ⍺⍺ ⍵[3]`), or to the right (`⍵[1] ⍺⍺ (⍵[2] ⍺⍺ ⍵[3])`), hence the names *fold-right* and *fold-left*. Unlike reduction, the result of *folding* an empty array is defined and equal to the identity element.

- *Scanning* a list `⍵` with a function `⍺⍺` is equivalent to mapping a fold over prefixes of a list. This definition of a scan has a $O(n^2)$ complexity regardless of the fold type. A more efficient, $O(n)$ definition of a *scan-left* exists (fold with intermediate steps).

- *Outer product* of function `⍺⍺` (`∘.⍺⍺`) and arrays `⍺` and `⍵` is the application of `⍺⍺` between every pair of elements from `⍺` and `⍵`. The resulting list has a depth given by `⍴,⍤⍴`.

- *Inner product* of functions `⍺⍺`, `⍵⍵` and lists `⍺` and `⍵` (`⍺⍺.⍵⍵`) is equivalent to folding the list obtained by putting `⍵⍵` between corresponding pairs of `⍺` and `⍵` with `⍺⍺`.

- A *higher-order function* is a function that takes another function as its argument.

- A *lambda expression* (in the context of KamilaLisp) is an anonymous function with static (lexical) scoping, meaning that it can see all the variables bound by its lexical ancestor, unlike to dynamic scoping implemented by older LISPs where a function can see all the variables bound by its caller(s). In both cases, the first bound variable found is used, allowing to shadow variables.

- A *dyad* is a two argument function.

- A *monad* is generally a single argument function. The book also uses it in a context of an abstract data constructor implementing the *bind* and *unit* functions.

- *Replicating* a list `⍵` according to list `⍺` is copying each element of `⍵` a given number of times (specified by the corresponding element of `⍺` or assumed to be zero). In KamilaLisp, `⍺` can be a scalar, in which case the contents of `⍵` are catenated to each other `⍺` times. If `⍵` is also a scalar, it's repeated `⍺` times to form a list.

- *Filtering* a list `⍵` with function `⍺⍺` is equivalent to mapping `⍺⍺` on every element of `⍵` (assuming `⍺⍺` returns a boolean value, i.e. `0` or `1`), and replicating `⍵` with the result of the mapping (`⍵/⍨⍺⍺¨⍵`).

- *Rotating* a list `⍵` by `⍺` elements is equivalent to dropping `⍺` elements from `⍵`, and joining them with `⍵` (dyadic `⍫`, or more illustratively, `(↓,↑)`).

- *Zipping* two arrays is the act of forming a list of pairs via the juxtaposition of corresponding elements from the given arrays.

- *Flattening* a list if the act of decreasing the list's depth by one level, enlisting all of the elements from the topmost sublists into a resultant list.

- *Mapping* a function `⍺⍺` over a list `⍵` is the act of processing each element of `⍵` with the function `⍺⍺` to produce a resultant list of equivalent length.
- *Partial application* is defined as fixing a number of arguments to a function, yielding an anonymous function of smaller arity.
- *Iteration* of a function over an argument is defined as evaluating the function over its result starting with the initial argument until a condition is satisfied. Sometimes the condition is numeric, or is defined as a dyadic function between the previous and current result (just `while`). A fixed point combinator could be implemented using partial application of deep equality (`⍣≡`). Iteration is faster and consumes less resources than recursion, assuming no tail call optimisation.
- *Function composition* (or `@`/`∘`, as KamilaLisp calls it) is taking an arbitrary amount of functions where each function operates on the results of the function before, except the last function, which gets all the arguments passed to the anonymous function yielded by composition.
- *Selfie* is a higher order function that duplicates the argument to the function it takes (e.g. `×⍨`, computes the square of its argument).
- *Commute* swaps the order of two arguments to a function. `⍺ f ⍵ ⟺ ⍵ f⍨ ⍺`.
- A *variadic function* is a function that takes an arbitrary number of arguments.
- *Currying* is the technique of translating the evaluation of a function that takes multiple arguments into evaluating a sequence of functions, each with a single argument.

Chapter 1

Initial considerations

This chapter discusses the basics of KamilaLisp. Throughout this book, the author will use the KamilaLisp interpreter to check and execute the declarations of a program one by one. The emphasis will be put on list processing and mathematical functions to form elementary understanding of the language.

1.1. Programs and variables

A KamilaLisp program is a sequence of declarations, which are executed in the order they are written. The first program presented in this book is shown below:

```
--> def a (+ (* 2 3) 2)
8
--> def b (* 5 a)
40
```

It consists of two declarations. The first declaration binds the identifier a to the integer 8, and the second declaration binds the identifier b to the integer 40, which follows from the intuitive understanding of the arithmetic operations. To the reader not accustomed with Lisp-like syntax, every element of the syntax tree that would otherwise be implicitly grouped by a language with usual arithmetical precedence rules is explicitly grouped by parentheses to form a list. The resulting values of variables can be determined as follows:

```
--> ?a
8
--> ?b
40
```

The question mark is a sign for the KamilaLisp interpreter not to evaluate the entire input as an expression, but rather, to query the value of what it refers to.

Every list besides the empty list (usually written as `()` or alternatively `nil`) has a *head* defined as the first element of it. When a Lisp program is evaluated, the *head* of the current list is assumed to be a callable value, while the rest of the list (also called the *tail*) is assumed to be a list of arguments.

Because the list of arguments to a function (the *tail*) is evaluated before its applied to the *head*, a perceptive reader could point out a potential issue - *How to introduce list literals in the code*? This question is indeed well-founded, since the list literal would be evaluated in order to pass it parameter to some callable object, hence the tail of the literal would be applied to its head, thus behaving undesirably and almost certainly raising an error. Every Lisp dialect addresses this issue in the same way using the quoting mechanism. Simply put, the quote prevents a list from being evaluated. To observe this behaviour, introduce two more functions called `car` and `cdr` to obtain respectively the *head* and *tail* of a list:

```
--> car '(1 2 3)
1
--> cdr '(1 2 3)
(2 3)
```

This may alert the observant reader once again - *What if I want to create a list out of a set of expressions*? This question is once again very relevant - because the quote stops a form from being evaluated, then surely the expressions inside of it will stay untouched too. The answer is simple - use the `tie` function as follows:

```
--> tie 1 2 3
(1 2 3)
--> tie (+ 2 2) (/ 6 3) (+ 2 3)
(4 2 5)
```

KamilaLisp follows scoping rules familiar from other programming languages, such as Scheme or C++ - *static scoping* (also called *lexical scoping*), where an attempt is initially made to resolve a variable in the current scope. If this approach fails, the variable is resolved in the scope of its lexical ancestors until either the interpreter finds an environment where the variable is bound, or raises an error regarding an unbound variable. Additionally, variables may be *shadowed*, as demonstrated below:

```
--> def my-list '(1 2 3)
(1 2 3)
--> car my-list
1
--> def my-list (cdr my-list)
(2 3)
--> car my-list
2
```

However, it is not possible to shadow pre-defined variables and functions in the global scope:

```
--> def car 5
RuntimeException thrown in thread 1dbd16a6:
        def can not shadow or redefine built-in bindings.
    at entity def  1:1
    at def primitive function
```

To fully exercise lexical scoping, the language needs to provide a way of binding names inside a specific block of code (unlike *def* which binds names in the global scope). This can be accomplished in a variety of ways, the most straight-forward one being the *let* construct. The *let* construct binds a list of name/value pairs and evaluates the body of the construct in the context of the newly created environment. The syntax of the *let* construct is demonstrated by the following example:

```
--> def a 5
5
--> def b 6
6
--> + a b
11
--> let ((a 10) (b 15)) (+ a b)
25
```

1.2. Functions and lambda expressions

Functions are the core component of KamilaLisp. They are first-class objects, which means that they can be passed argument to other functions, returned from functions and bound to names. The syntax of a function declaration is as follows:

```
--> defun square (x) (* x x)
(λ x . (* x x))
```

The function that has just been declared is called `square` and it takes one argument called `x`. The body of the function is the expression `* x x`. The function returns the value of its expression in an environment where its arguments are bounded, which in this particular scenario is naturally the square of the argument. Notice that when defining a monadic function, a pair of parentheses around its only argument's name can be omitted for brevity:

```
--> defun square x (* x x)
(λ x . (* x x))
```

Since `square` is now bound in the global scope, it can be applied to an argument. The code below binds the name `a` to the result of application of the number 5 to the function `square`:

```
--> def a (square 5)
25
```

Functions do not need to be named. They can be introduced in the code *anonymously* using the *lambda* construct, which opens up many new possibilities for structuring code. For example, the declaration of the function `square` can be rewritten as follows:

```
--> def square (lambda x (* x x))
(λ x . (* x x))
```

Furthermore, multivariate lambda expressions could serve as a substitute for the *let* construct:

```
--> let ((a 10) (b 15)) (+ a b)
25
--> (lambda (a b) (+ a b)) 10 15
25
```

Since functions are first-class in KamilaLisp, it is also possible to return them from functions and take them as arguments. The following example demonstrates these programming techniques using the *lambda* construct:

```
--> ; Returns a function that adds a given number to its argument.
--> ; The technique demonstrated is often called "currying".
--> defun make-adder (x) (lambda y (+ x y))
(λ x . (λ y . (+ x y)))
--> def add-5 (make-adder 5)
(λ y . (+ 5 y))
--> add-5 10
15
```

```
--> ; Returns a string explaining the value of a function at point.
--> defun explain (f x) (str:format "The value of f(x) for x={?x} is {f x}")
(λ f x . (str:format "The value of f(x) for x={?x} is {f x}"))
--> explain square 4.5
The value of f(x) for x=4.5 is 20.25
```

When implementing complex functions, it is often of particular interest to keep the partial results obtained during the execution of the function. This can be achieved by cascading the *let* construct or using the *let-seq* construct. The following examples implement a function that raises its only argument to the eighth power[1]:

```
--> defun p8 x (let-seq
...   (def y (* x x))
...   (def z (* y y))
...   (* z z))
(λ x . (let-seq (def y (* x x)) (def z (* y y)) (* z z)))
--> p8 4
65536
--> defun p8 x (let ((y (* x x))) (let ((z (* y y))) (* z z)))
(λ x . (let ((y (* x x)) (z (* y y))) (* z z)))
--> p8 4
65536
--> defun p8 x (let ((y (* x x)) (z (* y y))) (* z z))
(λ x . (let ((y (* x x)) (z (* y y))) (* z z)))
--> p8 4
65536
```

Notice that despite using `def`, `defun`, etc..., the *let-seq* construct does not create any bindings in the global scope - the bindings are always local to the block.

When a function call is performed in KamilaLisp, the *call stack* is modified in the process. The call stack is a data structure implemented inside of the interpreter that keeps track of the functions that are currently being executed. Application of a lambda function forces creation of a new *stack frame* which is subsequently created and pushed onto the stack. When the function returns, the stack frame is popped from the stack - this way, the interpreter knows where to return to after the function yields a value. When an exception is raised, the interpreter will present the user with a *stack trace*, which is a list of functions that were executed before the exception was raised, for example:

```
--> defun f x (/ 1 0) ; Oops! Division by zero!
(λ x . (/ 1 0))
--> defun g x (f x)
(λ x . (f x))
--> defun h x (g x)
(λ x . (g x))
--> h 10
ArithmeticException thrown in thread 5e82df6a:
    Division by zero
at entity h  1:1
at (λ x . (g x))  1:9
```

[1] Many reimplementations of common built-in functions mentioned in the book that are present in KamilaLisp are rather suboptimal and demonstrated only for the sake of completeness. The programmer is urged to use the optimised predefined routines when possible instead.

```
at entity g  1:12
at (λ x . (f x))  1:9
at entity f  1:12
at (λ x . (/ 1 ...))  1:9
at entity /  1:12
at / primitive function
```

1.3. Conditional expressions and comparisons

The comparison operators in KamilaLisp do not differ significantly from the ones present in other, perhaps more orthodox programming languages (such as C). It is worth noting that scalar[2] equality is checked using the `=` function, inequality is checked for using the `/=` function, while the `<=>` function is the so-called *three way comparison* operator[3], which returns -1, 0 or 1 respectively if the first argument is less than, equal to, or greater than the second argument.

KamilaLisp provides a number of conditional expressions which are used to control the flow of execution. The most basic one is the *if* construct which takes three arguments - a condition, an expression to be evaluated if the condition is true, and an expression to be evaluated if the condition is false. The syntax of the *if* construct is demonstrated by the following example:

```
--> defun my-abs x (if (< x 0) (- x) x)
(λ x . (if (< x 0) (- x) x))
--> my-abs -5
5
--> my-abs 5
5
--> my-abs 0
0
```

The *if* construct is a special case of the *cond* construct, which takes a list of pairs of conditions and expressions. The first condition that evaluates to a truthy value is used to evaluate the corresponding expression. The syntax of the *cond* construct is demonstrated by the following reimplementation of the three-way comparison operator:

```
--> defun compare (x y) (cond ((< x y) -1) ((> x y) 1) (0))
(λ x y . (cond ((< x y) -1) ((> x y) 1) (0)))
--> compare 5 10
-1
--> compare 10 5
1
--> compare 5 5
0
```

1.4. Recursive functions

Recursion is a powerful technique extensively used in functional programming. Its role in KamilaLisp is admittedly not as extensive as in other functional programming languages, since the language provides a number of other often more wieldy techniques for solving the same problems (e.g. using array

[2]Operating on scalar values, i.e. not vector, matrix or general tensor values.
[3]Also called the *spaceship* operator.

programming), however it is still worth mentioning that KamilaLisp provides a number of tools for writing recursive functions.

The following function implements the *factorial* function in a recursive manner:

```
--> defun factorial (n) (if (= n 0) 1 (* n (factorial (- n 1))))
(λ n . (if (= n 0) 1 (* n (factorial (- n 1)))))
--> factorial 5
120
```

Since KamilaLisp supports arbitrary precision numerical computation, the factorial function can be applied to arbitrarily large numbers. However, at some point this function will *overflow its call stack*, meaning that the number of recursive calls will exceed the maximum call stack size. This is a common problem with recursive functions and it can be solved by using *tail recursion*. Tail recursion is a special case of recursion, where the recursive call is the last expression in the function body. In this case, the stack frame of the current function can be reused for the recursive call, hence the stack will cease to grow uncontrollably. To make use of this technique, the factorial function needs to be altered in a way that the recursive call is the last expression in the function body:

```
--> defun factorial (n) (let-seq
...   (defun f' (n acc) (if (= n 0) acc (f' (- n 1) (* n acc))))
...   (f' n 1))
--> factorial 5
120
```

The factorial function defines a helper function that has an accumulator argument, which is used to store the intermediate results of the computation. The helper function is called recursively and the accumulator argument is updated with the result of the multiplication. The *let-seq* construct is used to define the helper function, so that it is not visible outside the factorial function. There is one more step to make this function tail-recursive: replace the self-referential call in the helper function:

```
--> defun factorial (n) (let-seq
...   (defun f' (n acc) (if (= n 0) acc (&0 (- n 1) (* n acc))))
...   (f' n 1))
--> factorial 5
120
```

The self-referential call was replaced by `&0` which is a reference to the current function. This is a special case of the `&` operator, which is used to refer to functions by nesting level in the source code, akin to de Bruijn indices. The `&0` operator refers to the current function, `&1` refers to the function (anonymous or named) that is the lexical ancestor the current function, and so on. This way, the factorial function no longer raises an error when applied to large numbers, since the stack does not grow beyond the maximum size:

```
--> factorial 1000
402387260077093773543702433923003985719374864210714
632543799910429938512398629020592044208486969404800
; [...]
000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000
```

Another closely related function to the factorial function that can be implemented using tail recursion is the power function. The following expression implements the power function in a simple recursive manner:

```
--> defun power (x n) (if (= n 0) 1 (* x (power x (- n 1))))
(λ x n . (if (= n 0) 1 (* x (power x (- n 1)))))
--> power 2 10
1024
```

Once again, the problem is that the stack will overflow when the power function is applied to large numbers. To prevent this and speed up the computation, the power function needs to be first transformed so that the recursive call is the last expression in the function body. Consider the following *trace* of the power function applied to the arguments `2` and `4`:

```
--> power 2 4
(* 2 (power 2 3))
(* 2 (* 2 (power 2 2)))
(* 2 (* 2 (* 2 (power 2 1))))
(* 2 (* 2 (* 2 (* 2 (power 2 0)))))
(* 2 (* 2 (* 2 2)))
(* 2 (* 2 4))
(* 2 8)
16
```

After transforming the function to use an accumulator:

```
--> defun power (x n) (let-seq
...   (defun f' (x n acc) (if
...       (= n 0)
...       acc
...       (f' x (- n 1) (* x acc))))
...   (f' x n 1))
```

When this implementation of the power function is applied to the same arguments as before, the following trace is produced:

```
--> power 2 4
(f' 2 4 1)
(f' 2 3 2)
(f' 2 2 4)
(f' 2 1 8)
(f' 2 0 16)
16
```

Notice, that the size of the expression *does not grow*, hence the function could in theory be rewritten as a simple loop with a constant stack usage requirement - a concept more familiar from imperative programming languages such as C or C++. The final improvement that needs to be applied is actually coaxing the interpreter to perform tail call optimisation using the self-referential call `&0`:

```
--> defun power (x n) ((
```

```
...    lambda (x n acc) (
...        if (= n 0) acc
...            (&0 x (- n 1) (* x acc)))) x n 1)
--> power 2 4
16
```

1.5. Function composition

Function composition is a a common core concept in functional programming languages, an emphasis on which is placed in KamilaLisp. Using the `@` operator, it is possible to compose two or more functions into a single function. An example follows:

```
--> defun f x (* x 2)
(λ x . (* x 2))
--> defun g x (+ x 1)
(λ x . (+ x 1))
--> f 2
4
--> g 2
3
--> f@g 2
6
```

The usefulness of function composition may be challenging to appreciate at first. In the end, `f@g x` ⇔ `f (g x)`, however, the function returned by the `@` operator does not have to be immediately applied - it can be bound to a name and used later. Without the composition operator, it would be necessary to introduce a *lambda* to achieve the same result as `f@g` - that is, `lambda x (f (g x))`. The difference between manually composing two functions to create a new function and using the `@` operator is how they treat arguments[4]. The `@` operator variant of the same expression does not refer to the arguments of the composed functions (and thus does not manually relay an indeterminate amount of them to the innermost function), creating the basic building block for *point-free programming*.

Of course, it is possible to compose arbitrarily many functions by using the `@` operator multiple times. The following example demonstrates the composition of three functions:

```
--> defun f x (* x 2)
(λ x . (* x 2))
--> defun g x (+ x 1)
(λ x . (+ x 1))
--> defun h x (- x 3)
(λ x . (- x 3))
--> f@g@h 2
0
```

Another form of function composition notoriously used in the APL family of programming languages is the so-called *fork* (the μ-recursive composition operator). Generally speaking, it is sometimes of special interest to preprocess the arguments using different functions (*reductees*), and then funnel the results into a single function (*reductor*). For instance, the arithmetic mean is defined as the quotient

[4]Function composition performed using the `@` operator does not require creating a new stack frame for the lambda function.

(*reductor*) of the sum and length of a list (*reductees*), while a palindrome is defined as a string for which the equality (*reductor*) between it and its reverse (*reductees*) holds.

To implement a sum function using a fork, it is necessary to define a function that sums the list beforehand using the `tally` function which returns the length of a list. A non-zero value is considered truthy, hence the following definition of the *sum* function:

```
--> defun sum (l) (if (tally l) (+ (car l) (sum (cdr l))) 0)
(λ l . (if (tally l) (+ (car l) (sum (cdr l))) 0))
--> sum '(1 6 2 3)
12
```

There are many alternative ways to implement it. One problem the reader may stumble upon is `= 'nil 'nil` returning `'nil`. This behaviour is reasoned by the fact that equality *vectorises* over lists[5], meaning that equality is tested element-wise. To determine structural equality, it is advised to use the `same` function. The following example demonstrates this behaviour:

```
--> = '(1 2 3) '(1 2 4)
(1 1 0)
--> same '(1 2 3) '(1 2 4)
0
--> same '(1 2 3) '(1 2 3)
0
```

Hence arguing for a simpler implementation of the *sum* function:

```
--> defun sum (l) (if (same l '()) 0 (+ (car l) (sum (cdr l))))
(λ l . (if (same l '()) 0 (+ (car l) (sum (cdr l)))))
--> sum '(1 6 2 3)
12
```

The arithmetic mean function is thusly introduced as follows:

```
--> [/ sum tally] '(1 6 2 3 4)
3.2
```

Naturally, it could also be bound to a name without applying the fork instantaneously:

```
--> def avg [/ sum tally]
[/ sum tally]
--> avg '(1 6 2 3 4)
3.2
```

The key merits of point-free programming are:

- The code is more concise.
- The code is often more readable.
- The code is faster, because there is no need to allocate a stack frame.

[5] A more specific way to phrase it would be to say that `=` is a *pervasive function*.

- The code does not need to bind any names.

The lack of argument naming gives the point-free paradigm a reputation of being obtuse, hence the humorously used epithet "pointless style"[6]. However, the point-free paradigm is not necessarily obscure. In fact, it is often more readable than the equivalent imperative code and favoured by many languages, such as Haskell, APL, J, PostScript, Forth, Factor, jq and the Unix shell[7].

1.6. Partial application and μ-recursive functions

Many functional programming languages such as OCaml and Haskell automatically *curry* functions by default, that is, allow applying functions to fewer arguments than they are defined to take. This is a useful feature, since it allows for the creation of new functions by binding some of the arguments of a function to a value. This mechanism is called *partial application*. KamilaLisp supports partial application of functions but it does not happen by default, as unlike OCaml and Haskell, KamilaLisp supports variadic functions. Recall the following example given earlier in the book:

```
--> ; Returns a function that adds a given number to its argument.
--> ; The technique demonstrated is often called "currying".
--> defun make-adder (x) (lambda y (+ x y))
(λ x . (λ y . (+ x y)))
--> def add-5 (make-adder 5)
(λ y . (+ 5 y))
--> add-5 10
15
```

Using partial application, the example can be rewritten as follows:

```
--> defun add-x n $(+ n)
(λ n . $(+ n))
--> def add-5 (add-x 5)
$(+ n)
--> add-5 6
11
```

Partial application is particularly useful in conjunction with various kinds of function composition, as it allows for the vast majority of functions that are defined in terms of other functions to be defined in a point-free manner. Another valuable example would be a function that returns the remainder of a number when divided by two.

```
--> defun mod-2 (n) (mod n 2)
(λ n . (mod n 2))
--> mod-2 10
0
--> mod-2 11
1
```

[6] http://hdl.handle.net/1822/2869

[7] The pipe operator is essentially point-free function composition. The Unix shell has more interesting properties from a category theoretic perspective, due to its extensive use of monads. The pipe operator is an implementation of `bind`, `cat` is in reality monadic `return`. Moreover, a pair of these operations, adding `<` and `>`, satisfy the monadic laws - `cat f | cmd` is the same as `cmd <f` (left-hand identity); `cmd | cat` is the same as `cmd` (right-hand identity); `c1 | (c2 | c3)` is the same as `(c1 | c2) | c3`.

It is not immediately clear how to define this function in a point-free manner, since partial application will apply the arguments in order as given, and the argument that needs to be partially applied here is actually the *second* argument to `mod`. This problem can be solved in a two ways - the first one is to use the commute operator `^` which reverses the argument order to a function:

```
--> def mod-2 $(^mod 2)
$['^mod, '2]
--> mod-2 5
1
--> mod-2 6
0
```

Another, perhaps more elegant solution is to use partial application placeholders:

```
--> def mod-2 $(mod _ 2)
$['mod, _, '2]
--> mod-2 5
1
--> mod-2 6
0
```

This way, the `_` placeholder indicates that the first argument given to the resulting function should be applied in its place. The placeholder can be used multiple times and it is required that all placeholder slots are filled when applying the already partially applied function.

Variadic functions are functions that take an arbitrary number of arguments. `+` is a good example of a variadic function, since when applied to more than two arguments it just sums all of them:

```
--> + 1 2 3 4 5
15
```

It is possible to define custom variadic functions in KamilaLisp using special argument syntax. To recall, `lambda (x y) (code)` takes two arguments - x and y. This function can be made variadic by prepending the *last argument* with an ellipsis - `lambda (x ...xs) (code)`. The last argument will be bound to a list of all the remaining arguments. This has an interesting property: while the aforementioned function takes *at least* one argument (because `...xs` can be empty), it can be modified to take any amount of arguments, even zero - `lambda ...xs (code)`. To demonstrate this behaviour, the following function will obtain the arithmetic average of all its arguments, but will refuse to be called with zero arguments:

```
--> defun avg (x ...xs) (/ (+ x (sum ...xs)) (+ 1 (tally ...xs)))
(λ x ...xs . (/ (+ x (sum ...xs)) (+ 1 (tally ...xs))))
--> avg 1 6 2 3 4
3.2
--> avg 5
5
--> avg
TypeError thrown in thread 15eb5ee5:
        Expected at least 1 arguments to `(λ x ...xs . (/ (+ x ...) ...))'.
    at entity avg  1:1
    at (λ x ...xs . (/ (+ x (sum ...xs)) ...))  1:11
```

Equipped with the power of variadic functions and recursion, the next natural step is to define μ-recursive functions. Consider the following *basic* μ-recursive functions:

- For all natural numbers i, k where $0 \leq i \leq k$, the projection function (sometimes also called the identity function when $i = 0$) is defined as $P_i^k(x_0, \ldots, x_k) = x_i$. The projection function is a built-in operator in KamilaLisp, where `#a` is equivalent to P_a^k.
- For each natural n and k, the constant function C_n^k is defined as $C_n^k(x_0, \ldots, x_k) = x_n$. This is easily implemented using the previously discussed projection function as `$(#0 n)`.
- For each natural n, the successor function S_n is defined as $S_n(x) = x + 1$. This is easily implemented using just partial application - `$(+ 1)`.

The μ-recursive composition operator (also called the substitution operator) defined for an m-ary function $h(x_0, \ldots, x_m)$ and exactly m n-ary functions $g_0(x_0, \ldots, x_n), \ldots, g_m(x_0, \ldots, x_n)$ as $h \circ (g_0, \ldots, g_m) = f$ where $f(x_0, \ldots, x_n) = h(g_0(x_0, \ldots, x_n), \ldots, g_m(x_0, \ldots, x_n))$. This definition is essentially equivalent to KamilaLisp *forks*[8] - `[h g_0 ... g_m] x_0 ... x_k`.

The μ-recursive primitive recursion operator $\rho(g, h) = f$ for k-ary function $g(x_0, \ldots, x_k)$, $k+2$-ary function $h(y, z, x_0, \ldots, x_k)$ and $k+1$-ary function f yields the following piecewise function:

$$f(a, x_0, \ldots, x_k) = \begin{cases} g(x_0, \ldots, x_k) & \text{if } x = 0 \\ h(a-1, f(a, x_0, \ldots, x_k), x_0, \ldots, x_k) & \text{if } x \neq 0 \end{cases}$$

The KamilaLisp implementation of this concept is slightly more involved, requiring the `lift` function to apply a function on a existing variadic parameter pack:

```
--> defun mu-prim-rec (g h) (
...     lambda (a ...xs) (
...         if (= a 0)
...             (lift g ...xs)
...             (lift $(h (- a 1) (lift $(mu-prim-rec g h) ...xs)) ...xs)))
```

The μ-recursive *minimization* operator is less demanding to implement. Intuitively, minimisation seeks, beginning the search from 0 and proceeding upwards, the smallest argument that causes the function to return zero; if there is no such argument, of if one encounters an argument for which f is not defined, then the search never terminates, and is not defined for the argument.

```
--> defun mu-min f (lambda ...xs (let-seq
...     (defun mu-min-iter i (if (lift f ...xs) (&0 (S i)) i))
...     (mu-min-iter 0)))
```

1.7. Iteration

Given a function f, the n-fold application of f to x is usually denoted in mathematics as $f^n(x)$. For instance, $f^0(x) = x$, $f^1(x) = f(x)$, and $f^2(x) = f(f(x))$. More generally, a recursive relation can be defined as $f^k(x) = f(f^k - 1(x))$ for $k \geq 1$. Given *an iteration* of a function $f^n(x)$, the function f is called the *step function*, n is called the order of iteration and x is called the *starting value*. In KamilaLisp, iteration is introduced using the `while` higher order function which, in the presently most useful form to the mathematical definition, takes three arguments - the starting value, the order of iteration and the step function.

[8] A general version of APL 3-trains; https://aplwiki.com/wiki/Train#3-trains

Consider the successor function `defun s x (+ x 1)`. To provide a basic example, addition of two numbers `+ a b` can be implemented as an iteration of order b of the successor function `s` with the initial value a:

```
--> defun add (a b) (while a b $(+ 1))
(λ a b . (while a b $(+ 1)))
--> add 3 4
7
```

Recall the power function and its previously discussed, tail-recursive definition:

```
--> defun power (x n) ((
...    lambda (x n acc) (if (= n 0) acc (&0 x (- n 1) (* x acc)))) x n 1)
(λ x n . ((λ (x n acc) (if (= n 0) acc (&0/syn x (- n 1) (* x acc)))) x n 1))
```

Since tail recursion is essentially equivalent to iteration, the power function may be implemented as an iteration of order $n - 1$ of the multiplication function `*` with the initial value x:

```
--> defun power (x n) (while x (- n 1) $(* x))
(λ x n . (while x (- n 1) $(* x)))
--> power 2 4
16
```

Iteration using `while` and other higher order functions is described later on in the book. As a tool, iteration is more suitable to certain problems over recursive (functional) or array programming approaches[9].

Of course, iteration with a fixed number of steps is the simplest form of iteration that covers many usages of the `for` loop from other more orthodox languages, such as C. However, iteration is often conditional and depends on some particular predicate. Consider the Collatz conjecture, one of the most famous unsolved problems in mathematics. The conjecture asks whether iterating this function as many times as needed will eventually reach the number 1:

$$f(n) = \begin{cases} \frac{n}{2} & \text{if } n \equiv 0 \pmod 2 \\ 3n+1 & \text{if } n \equiv 1 \pmod 2. \end{cases}$$

Providing a formal proof of the Collatz conjecture is beyond the scope of this book, but KamilaLisp makes it easy to test the conjecture for a finite set of numbers[10]. Start by implementing the function f in code and then use it in a recursive approach and a tail-recursive approach:

```
--> defun collatz (n) (if (= (mod n 2) 0) (/ n 2) (+ (* 3 n) 1))
(λ n . (if (= (mod n 2) 0) (/ n 2) (+ (* 3 n) 1)))
--> defun collatz-rec (n) (if (= n 1) 1 (collatz-rec (collatz n)))
(λ n . (if (= n 1) 1 (collatz-iter (collatz n))))
--> defun collatz-tail (n) (if (= n 1) 1 (&0 (collatz n)))
(λ n . (if (= n 1) 1 (&0 (collatz n))))
```

[9] For instance, computing the *Convex Hull* using the Graham scan algorithm, which has better asymptotic complexity than the more naive Jarvis scan as easily implemented in array fashion

[10] Even though the Collatz conjecture has been proven using computers for extremely large values of n, this is not enough evidence to claim that the Collatz function is likely true - consider the Polyá conjecture, for which the initial *disproof by counterexample* quoted a number estimated to be around $n = 1.845 \times 10^{361}$

After testing the functions for a few small numbers (say, $n < 1000$), it is clear that the conjecture holds for them. To rewrite this tail-recursive function using iteration, it is required to use the alternate definition of `while`. The second argument to `while` can be an integer as demonstrated previously, however it can also be a function that returns a boolean value. The iteration will continue as long as the function returns a truthy value. The new function `collatz-whl` will iterate the `collatz` function until n reaches 1:

```
--> defun collatz-whl (n) (while n $(/= 1) collatz)
(λ n . (while n $(/= 1) collatz))
```

Another way to use `while` is to omit the predicate argument and return a two element list of whether to continue iteration (true/false - yes/no) and the new value, however this approach is not demonstrated due to its lack of utility.

It might be of particular interest to determine the number of iterations and the numbers that have been reached in the process. This can be done using the `partial-while` function which, as the name suggests, iterates a function and yields a list of partial results that have been obtained before the final result:

```
--> defun collatz-list (n) (partial-while n $(/= 1) collatz)
(λ n . (partial-while n $(/= 1) collatz))
--> collatz-list 15
(46 23 70 35 106 53 160 80 40 20 10 5 16 8 4 2 1)
```

Another valuable comparison to be made between recursion, tail recursion and iteration is the implementation of Fibonacci numbers. Despite being implemented in KamilaLisp already as the function `fib`, consider the following reimplementation:

```
--> defun fibr (n) (if (< n 2) n (+ (fibr (- n 1)) (fibr (- n 2))))
(λ n . (if (< n 2) n (+ (fibr (- n 1)) (fibr (- n 2)))))
--> fibr 5
5
--> fibr 10
55
```

To derive a tail-recursive form of this function, it is necessary to use two accumulators:

```
--> defun fibt (n) ((
    ...    lambda (n a b) (if (= n 0) a (&0 (- n 1) b (+ a b)))) n 0 1)
(λ n . ((lambda (n a b) (if (= n 0) a (&0/syn (- n 1) b (+ a b)))) n 0 1))
--> fibt 10
55
```

The iterative version of this function trivially follows, however, let's **assume** that it is *not possible* to implement the iterative or tail-recursive version of this function - so the only implementation available a slow function that overflows the stack often. While in practice this is not the case, it is a good exercise to consider different ways in which the naive fibonacci function be improved.

Notice that the fibonacci function is rather expensive to compute, yet it takes just a number and returns just a number, so it is a good candidate for memoization. The memoization function `memo` takes a function and returns a memoized version of it. The memoized version of the function will store the results of previous calls and return them if the same arguments are passed to the function again. This is a very useful technique to improve the performance of functions that are expensive to compute and are called with the same arguments repeatedly via a technique akin to dynamic tabulation of a function. The memoized version of the fibonacci function is, to no surprise, created as follows:

```
--> def fibr (memo fibr)
memo$(λ n . (if (< n 2) n (+ (fibr (- n 1)) (fibr (- n 2)))))
--> fibr 10
55
--> fibr 20
6765
--> fibr 30
832040
--> fibr 50
12586269025
-->
```

The memoized version still has some issues (e.g. it will still overflow the stack if the cache is not big enough) but it is a good improvement over the original version. The built-in function `fib` in KamilaLisp uses the Binet formula to compute the Fibonacci numbers, which is a much faster method than the naive recursive approach.

One nuance that needs to be pointed out is that in functions and lambda expressions, names are updated ad-hoc and usually looked up when needed to allow the programmer to utilise some interesting concepts such as mutual recursion. This has a few consequences, for instance:

```
--> defun f (x) (+ x a)
(λ x . (+ x a))
--> def a 5
5
--> f 5
10
```

Another, more serious consequence is that the memoized version of the Fibonacci function will not quite work as expected if it is defined under a different name (e.g. as `def fibm (memo fibr)`). The function `fibm` will quickly return the result of `fibm 10` if this value was explicitly asked for before, but not for `fibm 11` - because the underlying function, `fibr`, is still recursing and the memoized version of it is simply not being used.

1.8. Exceptions

Exceptions are the main error handling mechanism in KamilaLisp, unlike in Haskell/Rust (which usually make use of optional types) or OCaml (which supplies exceptions alongside optional types). Exceptions usually unwind the call stack and exhibit worse performance characteristics in the exceptional case, but they are much easier to use and are more familiar to programmers coming from other languages. Additionally, they exhibit better performance in the non-exceptional case.

Using the `raise` function, the arithmetic average function could be rewritten to raise an exception if the list is empty:

```
--> defun avg (l) (if (same l 'nil) (raise "empty list") (/ (sum l) (tally l)))
(λ l . (if (same l 'nil) (raise "empty list") (/ (sum l) (tally l))))
--> avg '(1 2 3 4 5)
3
--> avg '()
RaiseError thrown in thread 6b927fb:
    empty list
```

```
at entity avg  1:1
at (λ l . (if (same l 'nil) ...))  1:11
at entity if  1:16
at if primitive function
at entity raise  1:34
at raise primitive function
```

The exception raised by `avg` and other kinds of exceptions (e.g. the one raised by `/ 1 0`) can be caught using the `try-catch` function. For instance, it can be employed to return the length of the error message instead of the average in the exceptional case, since the `error` variable is bound in the catch block supplying the error handler with the exception's message:

```
--> try-catch (avg '()) (tally error)
299
```

Of course, there is an extension to this system that allows the catch handler to distinguish different kinds of failures, as well as an extension to `raise` that lets the programmer supply a custom exception type. An example of this follows:

```
--> defun frobnicate (x) (cond
...     ((< x 10) (raise 'domain_error "Input too large."))
...     ((> x 10) (raise 'domain_error "Input too small."))
...     ((= x 0) (raise 'arithmetic_error "Division by zero."))
...     ((= x 1) (raise 'logic_error "Unimplemented for x=1."))
...     (/ 1 x))
```

The invocation and subsequent handling of the exception is as follows:

```
--> defun stringify-type error-id (cond
...    ((same error-id 'domain_error) "Domain")
...    ((same error-id 'arithmetic_error) "Arithmetic")
...    ((same error-id 'logic_error) "Logic"))
--> try-catch (frobnicate 0) (str:format
...    "Caught a {stringify-type error-id} error of length {tally error}.")
Caught a Domain error of length 357.
```

Chapter 2

Elementary data structures

This chapter will describe in detail the most common data structures and their implementations in KamilaLisp. Many data structures in KamilaLisp are defined in terms of each other - for instance, all the functions that generally operate on sets can also be used on lists, since sets constitute a special case of lists. This chapter will also describe elementary array programming in KamilaLisp, which is usually the preferred by many programmers way to solve complex problems quickly.

2.1. Basic list operations

Lists are one of the most important data structures used in functional programming. Every major programming language provides means of finite sequence storage and KamilaLisp is no different. A special emphasis is put on list and array processing, the basic building block of dataflow programming.

As mentioned before, every list besides the empty list contains a *head* (the first element, `car`) and a *tail* (the last element, `cdr`). The tail of a list is always another list, even if it is empty. The empty list literal is introduced in the code as `'nil` or `'()`.

Using `car` and `cdr` it is possible to define a basic, non-tail recursive function that yields the length of a list:

```
--> defun length l (if (same l 'nil) 0 (+ 1 (length (cdr l))))
(λ l . (if (same l 'nil) 0 (+ 1 (length (cdr l)))))
```

A generalised version of this function that handles scalar values, variadic application and strings is available as `tally`[1]:

```
--> tally '(1 2 3) '(4 5)
(3 2)
--> tally '(1 2 3 4 5)
5
--> tally "abcde"
5
--> tally 5
1
--> tally
0
```

Individual elements may be prepended to a list using the `cons` function:

[1] *tally* - to count or calculate something

```
--> cons 6 'nil
(6)
--> cons 5 (cons 6 'nil)
(5 6)
--> cons 1 '(2 3)
(1 2 3)
```

Hence, one could define a countdown function as follows:

```
--> defun countdown x (if x (cons x (countdown (- x 1))) '(0))
(λ x . (if x (cons x (countdown (- x 1))) '(0)))
--> countdown 5
(5 4 3 2 1 0)
```

Once again, a general result of this function is available in KamilaLisp as the `range` function:

```
--> range 5
(0 1 2 3 4)
--> range 5 10
(5 6 7 8 9)
--> range 10 5
(10 9 8 7 6)
--> range 5 -5
(5 4 3 2 1 0 -1 -2 -3 -4)
```

List concatenation in KamilaLisp is accomplished using the `append` function. The `append` function is of course variadic and accepts an empty parameter list:

```
--> append '(1 2 3) '(4 5)
(1 2 3 4 5)
--> append '(1 2 3) '(4 5) '(6 7)
(1 2 3 4 5 6 7)
--> append
'nil
--> append 'nil 'nil
'nil
--> append "Tomato" "sauce"
Tomatosauce
```

Prefixes and suffixes of lists may be extracted using the `take` and `drop` functions as follows:

```
--> take 3 '(1 2 3 4 5)
(1 2 3)
--> drop 3 '(1 2 3 4 5)
(4 5)
--> take 3 '(1 2 3)
(1 2 3)
--> drop 3 '(1 2 3)
--> take 3 'nil
```

```
[[]
 []
 []]
--> drop 3 'nil
--> take 5 '(1 2 3)
(1 2 3 0 0)
```

The `take` and `drop` functions also accept negative argument, which changes the direction of the operation:

```
--> take -3 "KamilaLisp is Fun"
Fun
```

More generally, *all* prefixes and suffixes of a list are extracted using the `prefixes` and `suffixes` functions:

```
--> prefixes '(1 2 3 4 5)
((1) (1 2) (1 2 3) (1 2 3 4) (1 2 3 4 5))
--> suffixes '(1 2 3 4 5)
((1 2 3 4 5) (2 3 4 5) (3 4 5) (4 5) (5))
--> suffixes "Lisp"
("Lisp" "isp" "sp" "p")
```

Going back to the `take` function, it is easy to notice that when the list is shorter than expected, the resultant list is simply padded with zeroes. This may not be the desired behaviour, thus a variant of `take` called `cycle` is provided. The `cycle` function takes a list and a number and returns a list of the same length as the number, where the elements are taken from the list in a cyclic manner:

```
--> cycle 5 '(1 2 3)
(1 2 3 1 2)
--> cycle 3 '(1 2 3)
(1 2 3)
--> cycle 2 '(1 2 3)
(1 2)
--> cycle 1 '(1 2 3)
(1)
--> cycle 0 '(1 2 3)
--> cycle -1 '(1 2 3)
RuntimeException thrown in thread 1dbd16a6:
        cycle: negative length
    at entity cycle  1:1
    at cycle primitive function
--> cycle 'nil 5
--> cycle "abc" 5
abcab
```

The `replicate` function ubiquitously used in APL and Haskell is also available in KamilaLisp, except its domain is extended to scalar values:

```
--> replicate 3 5
(5 5 5)
```

```
--> replicate 5 '(1 2 3)
(1 2 3 1 2 3 1 2 3 1 2 3 1 2 3)
--> replicate 5 "Kamila"
KamilaKamilaKamilaKamilaKamila
--> replicate 0 5
--> replicate 5 'nil
--> replicate '(1 2 3) '(4 5 6)
(4 5 5 6 6 6)
```

KamilaLisp also provides a few functions for altering the *order* of elements in a list. The `reverse` function reverses the order of elements in a list:

```
--> reverse '(1 2 3 4 5)
(5 4 3 2 1)
--> reverse "KamilaLisp"
psiLalimaK
```

The `rotate` function, as the name suggests, takes a list and a number and returns a list where the elements are rotated by the number of slots:

```
--> rotate 2 '(1 2 3 4 5)
(3 4 5 1 2)
--> rotate -2 '(1 2 3 4 5)
(4 5 1 2 3)
--> rotate -4 "KamilaLisp"
LispKamila
--> rotate 1 'nil
--> rotate 0 'nil
```

Finally, the `list:shuffle` function will take a list and return a list with the same elements, but in a random order:

```
--> list:shuffle '(1 2 3 4 5)
(2 5 3 1 4)
--> list:shuffle "KamilaLisp"
LpLsikmaia
```

2.2. Sorting, searching and indexing

KamilaLisp defines a special syntax for indexing into lists. The syntax is as follows:

```
--> def x '(1 5 2 3 4)
(1 5 2 3 4)
--> ?x$[0]
1
```

This syntax is very confusing when demonstrated in isolation. First, indexing returns a value without a function call involved, so it is mandatory to tell the interpreter that the intent is to obtain the value of an object, hence the `?` prefix in the REPL. It is possible to index a list using a list, as follows:

```
--> ?x$[1 3 4]
(5 3 4)
```

The indexing function loops over the list it has received and returns respectively the first, third and fourth items of a list, all tied together into a single array. This is a very powerful feature, as it allows for a very concise syntax for extracting elements from a list, applying permutations and even sorting, as demonstrated later in the book. Of course, indexing can also be done using an expression:

```
--> ?x$[random 5]
3
```

Another closely related functionality related to indexing is searching. The `index-of` takes a value and a list and returns the index of the first occurrence of the value in the list:

```
--> index-of 5 '(9 8 6 5 4 7 2 3)
3
--> index-of 5 '(9 8 6 4 7 2 3)
-1
```

2.3. Rank

KamilaLisp lists have *rank*, which is a measure of their nesting, usually interpreted in the context of how many dimensions they could have. For example, a doubly nested list can be interpreted as a matrix:

```
--> ?'((1 2) (3 4))
[[1 2]
 [3 4]]
```

Since a matrix usually has two axes, matrices (lists of lists of scalars) have rank 2. A vector is a list of scalar values, so it has rank 1. A scalar value has rank 0. The rank of an object can be computed using the `rank` function:

```
--> rank '((1 2) (3 4))
2
--> rank '(1 2 3 4)
1
--> rank 0
0
```

One interesting case to consider is a *ragged list* - a list whose elements have different ranks. For example, the following list is a ragged list:

```
--> ?'((1 2) (3 4) 5)
```

The first and second elements of the list have ranks 1 (vectors; lists of scalars), while the last element has rank 0 (a scalar). The rank of a ragged list is computed as if the maximum of ranks of a list was considered and the result is negated:

```
--> rank '((1 2) (3 4) 5)
-2
```

2.4. Elementary higher order functions

KamilaLisp provides a wide variety of higher order functions for manipulating lists. Many of them can be defined using recursion, however almost all of them are guaranteed to terminate, while recursion in its general case does not. The use of list processing functions that constitute the core of array programming is highly encouraged over recursion, because they tend to be more concise, easier to understand and less error prone.

The most used function is a built-in operator takes a function and a list and applies the function to each element of the list, yielding a list of the results. Define a successor function and map it over a list by prepending a single colon before the function name:

```
--> def s $(+ 1)
$['+, '1]
--> :s '(1 2 3 4 5)
(2 3 4 5 6)
```

The `map` function (which is the more familiar name of this construct, predominantly called that in Haskell and OCaml) has a few nuances that are worth mentioning. First, it is possible to apply it to a non-list argument and an empty list:

```
--> :s 5
(6)
--> :s 'nil
-->
```

The functor returned by `:` has the same arity as the function it is applied to, hence it is possible for it to act as a `zipWith` operation known from e.g. Haskell:

```
--> :+ '(1 2 3) '(4 5 6)
(5 7 9)
--> :+ '(1 2 3) '(4 5 6 7)
(5 7 9)
--> :+ '(1 2 3 4) '(4 5 6)
(5 7 9)
--> :+ '(1 1 1) '(2 2 2) '(3 3 3)
(6 6 6)
```

To present an example, the colon operator would be helpful in implementing a function to test whether its arguments are monotonically increasing or decreasing. The notation $x \leq y \leq z$ does not quite translate to KamilaLisp:

```
--> < 1 2 3
TypeError thrown in thread 9225652:
    2 arguments expected in application.
at entity <  1:1
at < primitive function
```

A function called `monotonic` can be defined to test whether a list of numbers is monotonically increasing or decreasing[2], based on the comparison function it takes argument:

[2]Monotonically decreasing means that every element of a sequence is smaller than the previous element.

```
--> defun monotonic (fn list) (same '(1) (unique (:fn list (cdr list))))
(λ fn list . (same '(1) (unique (:fn list (cdr list)))))
--> monotonic < '(1 2 3)
1
--> monotonic < '(1 4 3)
0
```

This particular example uses slightly inefficient logic (takes all unique elements of the mapping and checks if the resultant list is a singleton list[3] of `1`), while a more efficient implementation would use a higher order function to test whether all the elements of the resultant list are `1`, however this topic will be covered later on in the book.

Coming back to `map`, it is possible to specify *invariant arguments* to the function - the invariant arguments are constant arguments that are always supplied to the function being mapped, while other arguments over which `map` can iterate are changing:

```
--> :+ 5 '(5) '(1 2 3 4 5)
(11 12 13 14 15)
```

An important observation to be made is that it is possible to apply a function to a list of lists by stacking the comma operator multiple times also utilising the invariant arguments to write a function that forms tuples from the elements in a two-dimensional matrix:

```
--> def mat '((4 3) (3 4))
[[4 3]
 [3 4]]
--> ::cons 5 mat
(((5 4) (5 3)) ((5 3) (5 4)))
```

The colon operator may not be general enough to be suitable for all uses. For example, it may be desirable to create a *pervasive function* - a function which automatically applies itself to all the scalar values in a list. The built-in functions such as `+`, `-` or `ln` are pervasive by default, but for example the `reverse` function is not:

```
--> reverse '(("hi" "hello") ("kamila" "lisp"))
(("kamila" "lisp") ("hi" "hello"))
```

Since strings are generally considered scalar values by KamilaLisp (however, this is not the case in other array programming languages such as APL), applying it *on depth zero* yields the following results:

```
--> reverse%[0] '(("KamilaLisp" "is") "fun!")
(("psiLalimaK" "si") "!nuf")
```

The function `reverse` was ran on every object of rank zero of the list. If it is desirable reverse vectors (lists of scalars), the `reverse` function should be applied *on depth one*:

```
--> reverse%[1] '((1 2) 3 4)
((2 1) 3 4)
```

[3] A list containing only one element

To give another example, to reverse the rows of a list of matrices, the function should be applied *on depth two*:

```
--> reverse%[2] '(((1 2) (3 4)) ((5 6) (7 8)))
(((3 4) (1 2)) ((7 8) (5 6)))
```

Of course, since the depth operator is a *generalisation* of the colon operator (mapping), it is possible to use it to map a function over a list of lists. In this case, the depth specifier must be negative:

```
--> writeln%[-1] '("Hello" "world!")
```

The smaller negative number, the more times the map function is applied:

```
--> cons%[-2] mat 5
(((4 5) (3 5)) ((3 5) (4 5)))
--> ::cons mat 5
(((4 5) (3 5)) ((3 5) (4 5)))
```

An important thing to note is that the depth operator subtly differs from the colon operator in the variadic case. The colon operator will determine the shape of the result ad-hoc, regardless of argument order:

```
--> ::cons mat 5
(((4 5) (3 5)) ((3 5) (4 5)))
--> ::cons 5 mat
(((5 4) (5 3)) ((5 3) (5 4)))
```

The depth operator, however, will always infer the shape from its first argument, potentially leading to unexpected results:

```
--> cons%[-2] mat 5
(((4 5) (3 5)) ((3 5) (4 5)))
--> cons%[-2] 5 mat
[[5 4]]
```

This behaviour significantly differs from the behaviour of the depth operator in other languages, such as APL[4], where the depth operator is restricted to only two arguments, making it feasible to try dynamically determining the shape of the result. In KamilaLisp the shape is inferred from the first argument, since the operator's complexity would grow by a large magnitude as a result of it being a generalisation to an arbitrary amount of arguments. Additionally, the complex inferring rule in APL-like languages is not very useful in practice and leads to some design shortcomings.

To explore this topic further, it is necessary to demonstrate that the KamilaLisp depth operator accepts multiple depth values. For example, the following program will apply the function to objects of rank one extracted from the first array, and the objects obtained by descending once into the second array:

```
--> defun f (x y) (str:format "{?x}, {?y}")
(λ x y . (str:format "{?x}, {?y}"))
--> f%[1 -1] '((1 2) (3 4)) '(6 5 (4 3) 2 1)
("(1 2), 6" "(3 4), 5")
```

[4] https://aplwiki.com/wiki/Depth_(operator)

Notice that the depth operator makes an attempt to salvage the situation arising due to the fact that the lists are of different sizes by trimming the longer list to the size of the shorter one (which is not done by APL). To make a more fair comparison with APL, consider the following program instead:

```
--> f%[-2 0] '((1 2) 3) '((1 2) 3)
(("1, 1" "2, 2") "3, 3")
```

Since the depth operator simply extracts the objects of the specified rank from the arguments, it does not pay attention to the shape of other arguments, so scalars for the second argument to f can be extracted also in this case:

```
--> f%[-2 0] '((1 2) 3) '(1 2 3)
(("1, 1" "2, 2") "3, 3")
```

Since APL determines the shape in a more ”clever” (also way slower and more convoluted) way, this behaviour can not be achieved:

```
        ((1 2) 3) ({αω} Ö ¯2 0) (1 2) 3
┌──────────┬──┐
│┌──┬──┐   │3 3│
│|1 1|2 2|  |   |
│└──┴──┘   │   │
└──────────┴──┘
        ((1 2) 3) ({αω} Ö ¯2 0) 1 2 3
LENGTH ERROR
        ((1 2)3)({α ω} Ö ¯2 0)1 2 3
                ^
```

Finally, the depths list can be defined as a result of an expression:

```
--> f%[[tie + -] 1] '((1 2) (3 4)) '(6 5 (4 3) 2 1)
("(1 2), 6" "(3 4), 5")
```

A function similar to list mapping is `filter`. It takes a predicate and a list and returns a list of elements for which the predicate returned a truthy value.

```
--> filter (lambda x (> x 3)) '(1 2 3 4 5)
(4 5)
```

The relation between `filter` and `map` (the colon operator) can be observed by re-implementing one in terms of the other in the following way:

```
--> defun my-filter (pred lst) (replicate (= 1 (:pred lst)) lst)
(λ pred lst . (replicate (= 1 (:pred lst)) lst))
--> my-filter (lambda x (> x 3)) '(1 2 3 4 5)
(4 5)
```

`filter` allows for an elegant yet inefficient implementation[5] of a prime sieve. The sieve of Eratosthenes is an algorithm for finding all prime numbers up to a given limit. It works by iteratively marking the multiples of each prime number as composite. To implement this in KamilaLisp, it is needed to use recursion and two lists of numbers: one for the primes that have been already found, and one for the numbers that have not been classified yet.

Firstly, define the iteration step function that takes the prime and unclassified lists in a pair, adds the first element from the unclassified list to the prime list and removes all its multiples from the unclassified list:

```
--> defun step data (let-seq
    (def primes (car data))
    (def cands (car@cdr data))
    (case (same cands 'nil) (tie primes 'nil))
    (def current (car cands))
    (tie (cons current primes) (filter $(mod _ current) (cdr cands))))
```

This example of `let-seq` used the construct `case`, which is a special form of `if` that may be used only in `let-seq`. If the condition that directly follows `case` is true, then the further execution of the `let-seq`'s body is stopped and the value of the expression that follows the condition is returned. This is especially useful for terminating the computation when a special case is encountered. Using `while`, the iteration step function can be applied a finite amount of times to the initial pair of lists:

```
--> while (tie 'nil (range 2 50)) 1 step
((2) (3 5 7 9 11 13 15 17 19 21 23 25 27 29 31 33 35 37 39 41 43 45 47 49))
--> while (tie 'nil (range 2 50)) 5 step
((11 7 5 3 2) (13 17 19 23 29 31 37 41 43 47))
--> while (tie 'nil (range 2 50)) 10 step
((29 23 19 17 13 11 7 5 3 2) (31 37 41 43 47))
--> while (tie 'nil (range 2 50)) 15 step
((47 43 41 37 31 29 23 19 17 13 11 7 5 3 2) nil)
--> while (tie 'nil (range 2 50)) 20 step
((47 43 41 37 31 29 23 19 17 13 11 7 5 3 2) nil)
```

Since the step function *converges* to a value (eventually yields a value t such that $f(t) = t$), the `converge` function can be used to implement most of the sieve's logic now:

```
--> converge step (tie 'nil (range 2 100))
((97 89 83 79 73 71 67 61 59 53 47 43 41 37 31 29 23 19 17 13 11 7 5 3 2) nil)
```

To provide a final result, it is necessary to extract and reverse the prime list from the pair and wrap the invocation in a function:

```
--> defun sieve (n) (reverse@car@converge step (tie 'nil (range 2 n)))
(λ n . (reverse@car@converge step (tie 'nil (range 2 n))))
--> sieve 20
(2 3 5 7 11 13 17 19)
```

Ultimately, the function can be rewritten as follows to make it more concise and self-contained:

[5] Compared to the prime number-related primitive functions already supplied by KamilaLisp

```
defun sieve (n) (reverse@car@converge (
        lambda x (let-seq
            (def primes (car x))
            (def cands (car@cdr x))
            (case (same cands 'nil) (tie primes 'nil))
            (def current (car cands))
            (tie (cons current primes) (filter $(mod _ current) (cdr cands))))
    ) (tie 'nil (range 2 n)))
```

2.5. Folding and scanning

Folds and *scans* are higher-order functions that analyze a recursive data structure (usually a tree or list, however this section will focus only on folding lists) and through use of a given aggregation functor, combine the results of recursively processing its constituent parts, building up a return value. The difference between folds and scans is that the former return only the final result, while the latter return a list of intermediate results.

To demonstrate the simplest example of a fold, the sum of a list could be computed by folding it with the `+` function. Of course, the order of operation in this particular case does not matter since addition is commutative, however it is common to distinguish *left-associative folds* and *right-associative folds*. Imagine folding a list as inserting a dyadic function between its elements:

```
fold - (1 2 3 4 5) = 1 - 2 - 3 - 4 - 5
```

The sum can be parenthesised in two ways: as $(((1-2)-3)-4)-5$ or $1-(2-(3-(4-5)))$, thus former is a left-associative fold, while the latter is a right-associative fold.

Additionally, a question arises how should folds handle empty and singleton lists. Folds, in general case, return the identity element of the aggregation functor when the list is empty. When presented with a singleton list, the fold applies the function to the identity element and the only element of the list. Obviously, it is not possible to just query the identity element of the aggregating function, which is why the identity element of it is usually passed as an argument to fold. Sometimes it is desirable to fold a list with a function that does not have an identity element, in which case folding a list with a single element returns it and folding an empty list results in an error.

Of course, all of these variants of folding have their own names in KamilaLisp:

- `foldl` is a left-associative fold that takes an identity element and preprends it to the input list.
- `foldr` is a right-associative fold that takes an identity element and preprends it to the input list.
- `foldl1` is a left-associative fold that does not take an identity element and errors on empty lists.
- `foldr1` is a right-associative fold that does not take an identity element and errors on empty lists.

A sum from 1 to n can be naively computed using folds as follows:

```
--> foldl + 0 (range 1 100)
5050
```

The direction of the fold does not matter here. However, if presented with a non-associative functor, the direction of the fold matters. Consider the following example:

```
--> foldl - 0 (range 1 100)
-4950
--> foldr - 0 (range 1 100)
50
```

The first example essentially computes $((0 - 1) - 2) - \ldots$, while the second example computes $0 - (1 - (2 - \ldots))$.

The aggregating function is always a dyadic function. Depending on the direction of the fold, its arguments may be reversed. For example, in left-associative folds, the first argument is the accumulator and the second argument is the current element of the list. In right-associative folds, the first argument is the current element of the list and the second argument is the accumulator.

Folds make it possible to implement many familiar functions in a more elegant manner. Consider the arithmetic mean example introduced in the first chapter of the book:

```
--> defun sum (l) (if (same l '()) 0 (+ (car l) (sum (cdr l))))
(λ l . (if (same l '()) 0 (+ (car l) (sum (cdr l)))))
--> def mean [/ sum tally]
[/ sum tally]
```

The `mean` function can be rewritten as follows[6]:

```
--> def mean [/ $(foldl1 +) tally]
[/ foldl + 0 tally]
```

Folds, in a way, transcend the concept of tail recursion. A tail-recursive `foldl` function can be implemented in the following way:

```
--> defun my-foldl (f z l) (if (same l '()) z (&0 f (f z (car l)) (cdr l)))
(λ f z l . (if (same l 'nil) z (&0 f (f z (car l)) (cdr l))))
--> my-foldl - 0 (range 1 100)
-4950
```

It is also possible to define many concept using folds, such as `map` or `filter`:

```
--> defun my-map (f l) (foldr (lambda (x y) (cons (f x) y)) 'nil l)
(λ f l . (foldr (lambda (x y) (cons (f x) y)) 'nil l))
--> my-map $(+ 1) (range 1 10)
(2 3 4 5 6 7 8 9 10)
--> defun my-filter (f l) (foldr (lambda (x y) (if (f x) (cons x y) y)) 'nil l)
(λ f l . (foldr (lambda (x y) (if (f x) (cons x y) y)) 'nil l))
--> my-filter $(> 5) (range 1 10)
(1 2 3 4)
```

Perhaps more surprisingly, it is also possible to join two lists using `foldr`:

```
--> defun my-append (l1 l2) (foldr cons l2 l1)
(λ l1 l2 . (foldr cons l2 l1))
--> my-append (range 1 5) (range 6 10)
(1 2 3 4 5 6 7 8 9)
```

[6] In a very similar way to APL - compare `[/ $(foldl1 +) tally]` and `+/÷≢`.

This is due to the fact that `foldr` accepts an identity element, which in this function is the second list, and the list being folded as the first list - so in reality, what happens is:

```
my-append (range 1 5) (range 6 10)
= foldr cons (range 6 10) (range 1 5)
= cons 1 (cons 2 (cons 3 (cons 4 (cons 5 (range 6 10)))))
= (1 2 3 4 5 6 7 8 9)
```

Scans behave in a very similar way to folds, hence they share a common taxonomy. KamilaLisp supplies the following *scans*:

- `scanl` is a left-associative scan that takes an identity element and preprends it to the input list.
- `scanr` is a right-associative scan that takes an identity element and preprends it to the input list.
- `scanl1` is a left-associative scan that does not take an identity element and errors on empty lists.
- `scanr1` is a right-associative scan that does not take an identity element and errors on empty lists.

While `foldl1 +` will yield the sum of all the elements of a list, `scanl1 +` will yield a list of its partial sums (intermediate results of the fold):

```
--> foldl1 + (range 1 10)
45
--> scanl1 + (range 1 10)
(1 3 6 10 15 21 28 36 45)
```

Scans, especially in APL, are extensively used to implement certain kinds of problems. For example, the `prefixes` function is implemented in the following way:

```
--> def my-prefixes $(scanl1 append)
$['scanl1, 'append]
--> my-prefixes "hello!"
("h" "he" "hel" "hell" "hello" "hello!")
```

The `prefixes` functionality (which in KamilaLisp source code is implemented using methods isomorphic to a scan) can be also used to implement other, seemingly unrelated tools in mathematics. Consider the *Levi-Civita* symbol, extensively used in linear algebra, defined by the following explicit expression:

$$\varepsilon_{a_1 a_2 a_3 \ldots a_n} = \prod_{1 \leq i < j \leq n} \operatorname{sgn}(a_j - a_i)$$

A KamilaLisp implementation can be written as follows:

```
--> defun levi-civita (x) (if
...     ([not-same-elements unique #0] x)
...         0
...         (** -1 (foldl + 0 ([flatten@< #0 prefixes] x))))
```

A good way to simplify this code is to use the *backslash partitioning*. In essence, a pair of parentheses can sometimes be replaced with a single backslash in place of the opening parenthesis. The interpreter will replace the backslash with an open parenthesis of the type corresponding to the outer parenthesis and automatically double the next closing parenthesis of the same type that it encounters. To demonstrate, consider this version of the original code:

```
--> defun levi-civita x \
...    if ([not-same-elements unique #0] x)
...        0
...        \** -1 \foldl + 0 \[flatten@< #0 prefixes] x
```

To verify the code, generate a 2x2 Levi Civita symbol matrix:

```
--> levi-civita%[1] '(((0 0) (0 1)) ((1 0) (1 1)))
[[ 0 1]
 [-1 0]]
```

Another fascinating algorithm to implement using folds and scans is detecting spans of non-nested C-style comments. Consider the following C language declaration with the commented out parts underlined:

```
int x /* accumulator */ = 0 /* Store the amount of comments in the code. */ ;
      -----------------     ---------------------------------------------
```

Consider a function called `comments` with arguments `chars` and `str`, we find the closing comments as follows:

```
rotate -1 (reverse (find (reverse str) chars))
```

While opening comments are done without any further problems:

```
find str chars
```

The `find` function simply finds occurences of something in a string or list and returns a bit mask vector with the starting positions as follows:

```
--> find "an angry aardvark." "a"
(1 0 0 1 0 0 0 0 0 1 1 0 0 0 1 0 0 0)
```

Finally, the starting and ending comment symbols are put together into a single list as follows:

```
--> (defun comment (chars str) (:or
...     (find str chars)
...     (rotate -1 (reverse (find (reverse str) chars)))))
--> comment "/*" "This /* is */ an example"
(0 0 0 0 0 1 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0)
```

The final ingredient is connecting together bits and pieces of the problem is an application of `scanr1 /=` as follows:

```
--> defun comment (chars str) \rotate 1 \scanr1 /= \:or
...    (find str chars)
...    \rotate -1 \reverse@find (reverse str) chars
--> comment "/*" "This /* is */ an example"
(0 0 0 0 0 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0)
```

Finally, to accomplish the same underlining behaviour, one could use the following function:

```
--> defun display s (let-seq (def r " -"$[comment "/*" s]) (writeln s) r)
(λ s . (let-seq (def r " ¯"$[comment "/*" s]) (writeln s) r))
--> display "This /* is */ an example"
This /* is */ an example
     --------
```

2.6. Products and two-dimensional convolution

KamilaLisp supports two kinds of products known from mathematics: the inner (generalisation of *dot*) product and the outer (*cartesian*) product. The outer product of two lists is a list of all possible pairs of elements from the two lists:

```
--> [outer-product #0 #0] (range 1 4)
(((1 1) (1 2) (1 3)) ((2 1) (2 2) (2 3)) ((3 1) (3 2) (3 3)))
```

The dot product of two vectors is defined mathematically as follows:

$$x \cdot y = \sum_{i=1}^{n} x_i y_i$$

Inner product spaces generalize Euclidean vector spaces, in which the inner product is the dot product or scalar product of Cartesian coordinates. Consider the following example of implementing the dot product using the `inner-product` function:

```
--> inner-product + * '(1 2 3) '(4 5 6)
32
```

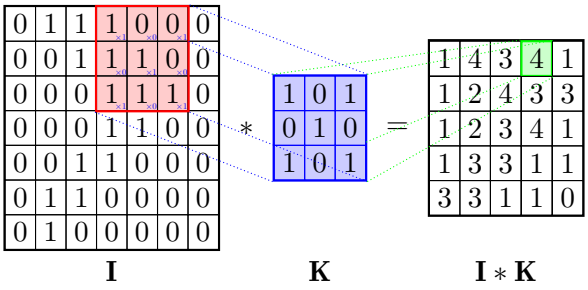
The first argument to `inner-product` is the summation operation, while the second argument is the multiplication operation. The third and further arguments are the two vectors to be multiplied. If only two arguments are provided, the `inner-product` function yields a function that performs the operation on arbitrarily many vectors.

To descend into technicalities, the `inner-product` function may be reimplemented using the `foldl1` function and `:` operator:

```
--> defun my-dot (f g ...xs) (foldl1 f (lift :g ...xs))
```

This implementation uses the `lift` function, which takes a list of arguments to be passed into a function and applies it to them. The equivalence between `lift` and usual function application is demonstrated as `lift f '(a1 a2 a3...) <=> f a1 a2 a3...`.

The inner product is particularly useful in the context of convolution. Throughout this section, it will be chiefly assumed that the convolution is two-dimensional. A step of convolution of two matrices **I** and **K** is illustrated as follows:



Notice that if one wanted to compute the value of the first cell of the matrix, this process would be complicated by the fact that the input matrix does not contain adequate data on the border. This difficulty is often called the *free boundary problem*. Throughout this section, it will be assumed that the data in the matrix is padded using the reflected boundary condition, which mirrors the data on the border. First, consider the following function `cell` that given a matrix, yields a function to query the value of a cell at a given position accounting for the reflected boundary condition:

```
defun cell (mat) \if (/= (rank mat) 2)
  (raise "Expected a matrix.")
  \lambda (x y) \let-seq
    (def h (tally mat))
    (def w (tally (car mat)))
    \cond
      ((and (< x 0) (< y 0)) mat$[- (+ y 1)]$[- (+ x 1)])
      ((and (< x 0) (< y h)) mat$[#0 y]$[- (+ x 1)])
      ((and (< x 0) (>= y h)) mat$[- (- (* 2 h) 1) y]$[- (+ x 1)])
      ((and (< x w) (< y 0)) mat$[- (+ y 1)]$[#0 x])
      ((and (< x w) (>= y h)) mat$[- (- (* 2 h) 1) y]$[#0 x])
      ((and (>= x w) (< y 0)) mat$[- (+ y 1)]$[- (- (* 2 w) 1) x])
      ((and (>= x w) (< y h)) mat$[#0 y]$[- (- (* 2 w) 1) x])
      ((and (>= x w) (>= y h)) mat$[- (- (* 2 h) 1) y]$[- (- (* 2 w) 1) x])
      \mat$[#0 y]$[#0 x]
```

The usages of the identity function `#0` stem from the fact that the indexing construct evaluates its argument if it is not a simple numeric list.

Define the function to compute the value of a two-dimensional convolution at a given position in the result matrix:

```
defun convolve-step (cell-mat kern x y) \let-seq
  (def kern-h \tally kern)
  (def kern-w \tally (car kern))
  (case (/= kern-h kern-w)
    \raise "Expected a square kernel.")
  (case (or (/= (mod kern-h 2) 1) (/= (mod kern-w 2) 1))
    \raise "Expected an odd-sized kernel.")
  (def kern-c \floor (/ kern-w 2))
  (def conv-range \flatten@outer-product
    (range (- x kern-c) \- (+ x kern-w) kern-c)
    (range (- y kern-c) \- (+ y kern-h) kern-c))
  (def conv-mat \:$(lift cell-mat) conv-range)
  \foldl + 0 \* (flatten conv-mat) \flatten kern
```

Now, define the `convolution-2d` function that takes argument the input matrix and the kernel matrix and yields the convolution of the two matrices using the `parallel-map-idx` function which behaves comparably to `$:` (the parallel variant of `:`), except it also passes the index in the list to the function:

```
defun convolve (mat kern) \let-seq
  (def cell-mat (cell mat))
  \parallel-map-idx (lambda (y row)
    \parallel-map-idx (lambda (x _e)
      \convolve-step cell-mat kern x y) row) mat
```

Finally, test the convolution on a simple example:

```
--> def mat '((1 2 3) (4 5 6) (7 8 9))
--> def kern '((0.11 0.11 0.11) (0.11 0.11 0.11) (0.11 0.11 0.11))
--> convolve mat kern
[[4.95 5.61 5.61]
 [6.93 7.59 7.59]
 [6.93 7.59 7.59]]
```

To implement convolution on RGBA images, it is necessary to split the color value into the separate channels. This can be accomplished using the `bit:unpack` function. Merging RGBA values together into a single integer is done using the `bit:pack` function:

```
--> bit:unpack 37126378 '(0 8) '(8 16) '(16 24) '(24 32)
(234 128 54 2)
--> bit:pack '(0 8 234) '(8 16 128) '(16 24 54) '(24 32 2)
37126378
```

Define a function that takes RGB image data (two-dimensional integer matrix), splits the color channels and applies a convolution kernel to each channel, then fuses the results together.

```
defun convolve-rgb (img kern) (let-seq
  ; Extract the R, G and B channels of the image data.
  (def r \$($(^/ 255)@bit:unpack _ '(0 8))%[0] img)
  (def g \$($(^/ 255)@bit:unpack _ '(8 16))%[0] img)
  (def b \$($(^/ 255)@bit:unpack _ '(16 24))%[0] img)
  ; Convolve each channel and convert back to integer values.
  (defun quantize x (cond ((< x 0) 0) ((> x 1) 255) ((round@* x 255))))
  (def r \quantize%[0] \convolve r kern)
  (def g \quantize%[0] \convolve g kern)
  (def b \quantize%[0] \convolve b kern)
  ; Merge the channels back together.
  ((lambda (r g b)
    (bit:pack
      (tie 0 8 r)
      (tie 8 16 g)
      (tie 16 24 b)
      (tie 24 32 255)))%[0] r g b))
```

Since the convolution code has become more complex, it should be saved to a file now and imported as a library. Declare public symbols that should be visible in the global scope using the `public:` prefix:

```
(defun cell (mat) \if (/= (rank mat) 2)
  (raise "Expected a matrix.")
  \lambda (x y) \let-seq
    (def h (tally mat))
    (def w (tally (car mat)))
    \cond
      ((and (< x 0) (< y 0)) mat$[- (+ y 1)]$[- (+ x 1)])
      ((and (< x 0) (< y h)) mat$[#0 y]$[- (+ x 1)])
      ((and (< x 0) (>= y h)) mat$[- (- (* 2 h) 1) y]$[- (+ x 1)])
      ((and (< x w) (< y 0)) mat$[- (+ y 1)]$[#0 x])
      ((and (< x w) (>= y h)) mat$[- (- (* 2 h) 1) y]$[#0 x])
      ((and (>= x w) (< y 0)) mat$[- (+ y 1)]$[- (- (* 2 w) 1) x])
      ((and (>= x w) (< y h)) mat$[#0 y]$[- (- (* 2 w) 1) x])
      ((and (>= x w) (>= y h)) mat$[- (- (* 2 h) 1) y]$[- (- (* 2 w) 1) x])
      \mat$[#0 y]$[#0 x])

(defun convolve-step (cell-mat kern x y) \let-seq
  (def kern-h \tally kern)
  (def kern-w \tally (car kern))
  (case (/= kern-h kern-w)
    \raise "Expected a square kernel.")
  (case (or (/= (mod kern-h 2) 1) (/= (mod kern-w 2) 1))
    \raise "Expected an odd-sized kernel.")
  (def kern-c \floor (/ kern-w 2))
  (def conv-range \flatten@outer-product
    (range (- x kern-c) \- (+ x kern-w) kern-c)
    (range (- y kern-c) \- (+ y kern-h) kern-c))
  (def conv-mat \:$(lift cell-mat) conv-range)
  \foldl + 0 \* (flatten conv-mat) \flatten kern)

(defun public:convolve (mat kern) \let-seq
  (def cell-mat (cell mat))
  \parallel-map-idx (lambda (y row)
    \parallel-map-idx (lambda (x _e)
      \convolve-step cell-mat kern x y) row) mat)

(defun public:convolve-rgb (img kern) (let-seq
  ; Extract the R, G and B channels of the image data.
  (def r \$($(^/ 255)@bit:unpack _ '(0 8))%[0] img)
  (def g \$($(^/ 255)@bit:unpack _ '(8 16))%[0] img)
  (def b \$($(^/ 255)@bit:unpack _ '(16 24))%[0] img)
  ; Convolve each channel and convert back to integer values.
  (defun quantize x (cond ((< x 0) 0) ((> x 1) 255) ((round@* x 255))))
  (def r \quantize%[0] \public:convolve r kern)
  (def g \quantize%[0] \public:convolve g kern)
  (def b \quantize%[0] \public:convolve b kern)
  ; Merge the channels back together.
  ((lambda (r g b)
    (bit:pack
```

```
        (tie 0 8 r)
        (tie 8 16 g)
        (tie 16 24 b)
        (tie 24 32 255)))%[0] r g b)))

"OK"
```

Finally, test the box blur on an image. Consider the following *original*, 256x256 RGB image:

Figure 2.1: peppers.jpg

Decrease numerical precision to speed up the computation. Create a 3x3 box blur kernel and apply it to the image:

```
--> import "convolution.lisp"
OK
--> let ((fr 10)) (img:write "peppers-blurry.jpg" (convolve-rgb
...     (img:read "peppers.jpg")
...     (* (/ 9) '((1 1 1) (1 1 1) (1 1 1)))))
peppers-blurry.jpg
```

Other interesting operations to check are the Gaussian blur and edge detection:

```
--> import "convolution.lisp"
OK
--> let ((fr 10)) (img:write "peppers-gauss.jpg" (convolve-rgb
...     (img:read "peppers.jpg")
...     (* (/ 16) '((1 2 1) (2 4 2) (1 2 1)))))
peppers-gauss.jpg
--> let ((fr 10)) (img:write "peppers-edge.jpg" (convolve-rgb
...     (img:read "peppers.jpg")
...     '((-1 -1 -1) (-1 8 -1) (-1 -1 -1))))
peppers-edge.jpg
```

(a) peppers-blurry.jpg (b) peppers-gauss.jpg (c) peppers-edge.jpg

To cover a different, related topic, consider the problem of matrix multiplication[7]. Generally speaking, matrix multiplication involves a series of dot products between the rows of the first matrix and the columns of the second matrix. For example, consider the following two matrices:

```
--> def A '((1 3 2 0) (2 1 0 1) (4 0 0 2))
[[1 3 2 0]
 [2 1 0 1]
 [4 0 0 2]]
--> def B '((4 1) (0 3) (0 2) (2 0))
[[4 1]
 [0 3]
 [0 2]
 [2 0]]
```

Start by transposing the second matrix and taking the outer product with `*`:

```
--> def B* (matrix:transpose B)
[[4 0 0 2]
 [1 3 2 0]]
--> def C (outer-product * A B*)
(((4 0 0 0) (1 9 4 0)) ((8 0 0 2) (2 3 0 0)) ((16 0 0 4) (4 0 0 0)))
```

The only thing left is to sum each simple vector of the result, which is done as follows:

```
--> def D ($(foldl + 0)%[1] C)
[[ 4 14]
 [10  5]
 [20  4]]
```

2.7. Searching and partitioning

2.8. Sorting and permutations

2.9. Pattern matching

2.10. Using glyphs

[7] In APL and other programming languages, it is possible to compute the dot product, perform matrix or tensor multiplication using the same construct - `+.×`.

Chapter 3

Functional data structures

Chapter 4

Mathematical programming

Chapter 5

Programming language theory

Appendix A - primitive functions